

Assignment 2: Allocator

For this assignment you will implement your own heap allocator similar to `ptmalloc2`, `jemalloc`, `tcmalloc`, and many others. These allocators are the underlying code of `malloc`. Heap allocators request chunks of memory from the operating system and place several (small) object inside these. Using the `free` call these memory objects can be freed up again, allowing for reuse by future `malloc` calls. Important performance considerations of a heap allocator include being fast, but also to reduce memory fragmentation.

Your allocator must implement its own `malloc`, `free` and `realloc` functions, and may not use the standard library versions of these functions. Your allocator may only use the `brk(2)` and `sbrk(2)` functions, which ask the kernel for more heap space.¹

Description of the functions to implement

You should implement `mymalloc`, `mycalloc`, `myfree`, and `myrealloc` as described in the Linux man pages.² These functions should behave exactly as specified by their man-page description, although you can ignore their *Notes* section as these are implementation-specific. Your allocations (i.e., the pointer returned by `mymalloc`) should be aligned to `sizeof(long)` bytes.

Your allocator may not place any restrictions on the maximum amount of memory supported or the maximum number of objects allocated. For example, your allocator should scale regardless of whether the maximum `brk` size is 64KB or 1TB.

Grading

This assignment is individual; you are not allowed to work in teams. Submissions should be made to the submission system before the deadline. Multiple submissions are encouraged to evaluate your submission on our system. Our system may differ from your local system (e.g., compiler version); points are only given for features that work on our system.

Your grade will be 1 if you did not submit your work on time, has an invalid format, or has errors during compilation.

If your submission is valid (on time, in correct format and compiles), your grade starts from 0, and the following tests determine your grade (in no particular order):

- +1.0pt if you make a valid submission that compiles.
- +1.0pt if your `malloc` returns a valid pointer to a new heap object. **Required**
- +0.5pt if your `calloc` returns a valid new heap pointer to zero-initialized memory.
- +2.0pt if a region of memory can be reused after freeing it with `free`. **Required**
- +1.0pt if `realloc` behaves as described on its man-page and only allocates a new object when needed.
- +1.0pt if your allocator batches `brk` calls, i.e., it does not need to request memory from the kernel for every allocation.
- +2.0pt if your amortized overhead per allocation is on average 8 bytes or less.
- +0.5pt if your allocator tries to optimize for locality (reuse recently freed memory).
- +1.0pt if your allocator gives back memory to the kernel (using `brk`) when a large portion of the allocated memory has been freed up.
- +1.0pt if your design does not use in-band metadata.
- +2.0pt if your allocation functions work correctly without the `my` prefix too (see *Notes* below).
- -2.0pt if your allocator cannot scale with the maximum `brk` size.

- -1.0pt if gcc -Wall -Wextra reports warnings when compiling your code.
- -1.0pt if your source files are not neatly indented or formatted.

If you do not implement an item marked with **Required** you cannot obtain any further points. This means you need to implement at least a simple allocator that can do malloc and free with reuse.

The grade will be maximized at 10, so you do not need to implement all features to get a top grade. Some features might be mutually exclusive with each other, depending on your allocator design.

Note: Your allocator will be evaluated largely automatically. This means features only get a positive grade if they work perfectly, and there will be no half grade for "effort".

Evaluation environment

For setting up a local development environment, refer to the setup document. In short, on Linux you should install build-essential python3 python2, on Windows you should use WSL2, and on macOS you should use Docker.

To test your implementation, the file test_framework/tests.c contains a number of (automated) test cases that evaluate the different aspects of your allocator. It can be invoked manually via ./test <test name>. Running make check (or make docker-check) will run all test cases, and additionally check your work for other errors that would lead to deducted points during grading.

Additionally you should test your work on our server. Remember to try this as often as you like, as your local environment may be different than ours. Points are only awarded based on what works on our server. The final submission before the deadline is used for grading.

Attempts to exploit, bypass or cheat the infrastructure and automated grading system will result in a 1 for this assignment.

Notes

- While you can edit the test framework locally to debug issues, you should not modify alloc.h or any file in test_framework/. During submission and grading any modifications made to these files will be thrown away.
- If you add definitions for malloc etc. to your alloc.c, you should also keep the original set of my functions for grading. Sample code that makes enables these functions is included in the skeleton alloc.c.
- If you have added support for replacing the system allocator (i.e., by adding non my prefixed functions) you can use your allocator for any existing program on your system. You can do this by prefixing any command with LD_PRELOAD=/path/to/libmyalloc.so. For example, LD_PRELOAD=./libmyalloc.so ls will run ls with your allocator.
- Calling your functions malloc instead of mymalloc not only redirects all calls inside **your** code to your malloc, but will also cause all internal libc calls to go to your allocator instead of the built-in libc malloc. Many libc functions, such as printf, internally make calls to malloc, and as such using printf inside your allocation code would cause an infinite loop. Therefore we prefix our allocator functions with my in this assignment.

1 Normal heap allocators may also use mmap to request memory from the kernel. For this assignment you should only use brk (or sbrk).

2 <https://linux.die.net/man/3/malloc>